Object-oriented construction of a multigrid electronic-structure code with Fortran 90

Yong-Hoon Kim,[1*] In-Ho Lee,[2] and Richard M. Martin[1,3]

[1] *Department of Physics, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*

[2] *School of Physics, Korea Institute for Advanced Study, Cheongryangri-dong, Dongdaemun-gu, Seoul 130-012, Korea*

[3] *Material Research Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*

(October 7, 1999)

Abstract

We describe the object-oriented implementation of a higher-order finite-difference density-functional code in Fortran 90. Object-oriented models of grid and related objects are constructed and employed for the implementation of an efficient one-way multigrid method we have recently proposed for the density-functional electronic-structure calculations. Detailed analysis of performance and strategy of the one-way multigrid scheme will be presented.

PACS: 02.70.-c,02.70.Bf,71.15.-m,71.15.Mb

Keywords: Electronic structure; Density-functional theory; Multigrid; Object-oriented programming; Fortran 90; Scientific computing

I. INTRODUCTION

In recent years, the usefulness of the real-space technique based on three-dimensional uniform grid and accurate forms of finite-difference formula has been demonstrated for the electronic-structure calculations [1–6]. Since all the computations are performed in real space, the major part of the calculations are local operations, which makes the algorithm easily parallelized and implementation of order-N algorithm [7] transparent. This is in contrast to plane wave method methods which rely upon global fast Fourier transforms. Furthermore, since the Laplacians and potential-wave functions multiplications are respectively evaluated by the finite-difference operation on the wave functions and a simple

**Corresponding author*

Address: Department of Physics, 1110 W. Green St., Urbana, IL 61801-3080, USA

E-mail: yhoon@physics.uiuc.edu

Telephone number: (217) 244-0391

Fax number: (217) 333-9819

one-dimensional vector multiplications on the fly, explicit storage of the Hamiltonian matrix elements can be avoided and the matrix diagonalization can be efficiently performed by iterative diagonalization methods such as conjugate gradient method.

Although there has been great emphasis on the algorithmic developments in the real-space electronic-structure calculation schemes, we feel that the issue of the code construction and organization has been relatively neglected. Recently, we have proposed an efficient and easy-to-implement one-way multigrid algorithm which results in significant enhancement in computation speed for grid-based iterative electronic-structure calculations [8]. In spite of the advantage of the multigrid in general [9,10,12], it requires complications of coding and organization of data structure because it involves several grid levels and data transfer between them. In such a situation, we found the modern programming construction paradigm, object-oriented[1] programming [13] can be useful. Our original code was written in Fortran 90 [14] with heavy recycling of our previous planewave codes written in Fortran 77, in conventional non-object-oriented programming style, with the purpose of typical scientific programming, namely the quick implementation of a physical idea. Although we tried to keep the code maintainable and clear, it quickly became long and messy with each addition of function and implementation of an idea. We recognized that the complexity of the conventional programming style is an obstacle, or at least a complication, especially for the implementation of multigrid. Hence, we restructured the code by introducing object-oriented modeling concepts, and in this work we will report our experience of this transition and the implementation of the multigrid method with the newly designed code. Although object-oriented programming should be most straightforward in object-oriented languages such as C++, it is in principle also possible in non-object-oriented languages [13], and especially relatively easy with Fortran 90 which supports many ingredients of object-oriented coding. Object-oriented scientific programming using Fortran 90 has been of much interest in recent years [15–19], and this paper will add additional information to this discussion. In addition, since grid-based simulations are common in other scientific and engineering computations, we expect our work is a helpful guide for the code construction in those fields.

The outline of the current paper is as follows. In Sec. II, we first describe the electronic-structure calculation scheme within the Kohn-Sham density-functional theory, and our methodology based on the higher-order finite difference formulation. In Sec. III, we review the key concepts of object-oriented methodologies, and describe the introduction of object-oriented concepts into our grid-based program written in Fortran 90. In Sec. IV, we briefly review the multigrid theory, and describe the one-way multigrid scheme which we have recently proposed [8]. Implementation of the one-way multigrid method is discussed, and especially the simplification induced by the object-oriented design is emphasized. The impressive enhancement of computational efficiency due to the introduction of the one-way multigrid method has been demonstrated in our previous publication, and here more detailed analysis of performance test and multigrid strategy will be reported. The current work will be summarized in Sec. V.

[1]Although "object-orientation" is both a language feature and a design methodology, this paper is primarily concerned with the design aspect.

II. HIGHER-ORDER FINITE-DIFFERENCE KOHN-SHAM ELECTRONIC-STRUCTURE CALCULATION METHOD

The (spin-dependent) Kohn-Sham (KS) density-functional theory (DFT) [20] is an independent-electron theory in which one obtains the single-particle wave functions $\psi_{\sigma \mathrm{i}}(\mathbf{r})$ for spin channel $\sigma =\uparrow, \downarrow$ and eigenvalues $\epsilon_{\sigma \mathrm{i}}$ by solving KS equations (Hartree atomic units are used throughout the paper)

$$\Big[-\frac{1}{2}\nabla^2 + V^{\mathrm{KS}}_{\mathrm{eff},\sigma}(\mathbf{r})\Big]\psi_{\sigma \mathrm{i}}(\mathbf{r}) = \epsilon_{\sigma \mathrm{i}}\psi_{\sigma \mathrm{i}}(\mathbf{r}), \tag{1}$$

with the spin density

$$n_\sigma(\mathbf{r}) = \sum_{i=1}^{N_\sigma} f_{\sigma \mathrm{i}}|\psi_{\sigma \mathrm{i}}(\mathbf{r})|^2, \tag{2}$$

where N_σ is the number of occupied σ spin orbitals. The effective KS potential $V^{\mathrm{KS}}_{\mathrm{eff},\sigma}(\mathbf{r})$ is composed of external, Hartree, and exchange-correlation contributions,

$$V^{\mathrm{KS}}_{\mathrm{eff},\sigma}(\mathbf{r}) = V_{\mathrm{ext}}(\mathbf{r}) + V_{\mathrm{H}}(\mathbf{r}) + V_{\mathrm{xc},\sigma}(\mathbf{r}), \tag{3}$$

among which $V_{\mathrm{H}}(\mathbf{r})$ and $V_{\mathrm{xc},\sigma}(\mathbf{r})$ depend on the charge density (and wave functions for orbital-dependent $V_{\mathrm{xc},\sigma}(\mathbf{r})$ [4]), hence Eqs. (1),(2), and (3) form a self-consistent system of equations. In the usual local density approximation, $V_{\mathrm{xc},\sigma}(\mathbf{r})$ is calculated inexpensively, hence the solution of KS equations [Eq. (1)] and generation of $V_{\mathrm{H}}(\mathbf{r})$ comprise main parts of calculations. For localized systems, $V_{\mathrm{H}}(\mathbf{r})$ are typically obtained by solving the Poisson equation

$$\nabla^2 V_{\mathrm{H}}(\mathbf{r}) = -4\pi n(\mathbf{r}). \tag{4}$$

where $n(\mathbf{r})$ is the total charge density $n(\mathbf{r}) = n_\downarrow(\mathbf{r}) + n_\uparrow(\mathbf{r})$.

In our higher-order finite-difference real-space formulation [1,3,4], we discretize both KS and Poisson equations on a three-dimensional uniform grid (with grid spacing h) with a higher-order finite difference representation (with finite-difference order N)

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}f(x) = \sum_{j=-N}^{N} C_j f(x+jh) + O(h^{2N+2}), \tag{5}$$

where $\{C_j\}$ are constants calculated by the algorithm of Ref. [21]. KS equations have been solved by the preconditioned conjugate gradient (CG) method [22,23] supplemented by subspace diagonalizations with *localized*[2] wavefunctions. The Hartree potential has been obtained from the solution of Poisson equation [Eq. (4)] on the *entire* simulation box, by first generating boundary values with multipole expansion, and then propagating solutions inside of the box with the combination of the low-order finite-difference (N=1) fast Fourier

[2]Vanishing boundary condition is used for wavefunctions. See Sec. III B.

transform method and the higher-order finite-difference (typically N=5) preconditioned CG method. At each self-consistent step we generate a new input Hartree-exchange-correlation potential using the simple linear mixing of output and input potentials. The computational procedure is summarized in Fig. 1. Further details of the computation method can be found in Ref. [3,4].

III. FORTRAN 90 IMPLEMENTATION OF OBJECT-ORIENTED CONCEPTS

A. Object-oriented programming

Roughly speaking, an object consists of a set of operations on some hidden data. Following Rumbaugh *et al.* [13], the key components of the object-oriented approach are:

1. *Identity* – data is quantized into discrete, distinguishable objects.
2. *Classification* – objects with the same data structure and behavior are grouped into a class.
3. *Polymorphism* – the same operation may behave differently on different classes.
4. *Inheritance* – sharing of data structures and behaviors among classes based on a hierarchical relationship.

In addition, there are several themes underlying object-oriented technology which give corresponding benefits:

- *Abstraction* – focusing on the essential, inherent aspects of an entity enhances the understanding of the problem itself, and preserves the freedom to make decisions as long as possible by avoiding premature commitments to details.
- *Encapsulation* – separating the external aspects of an object from the internal implementation details of the object prevents a program from becoming so interdependent that a small change has massive ripple effects.
- *Combining data and behavior* – keeping data structure hierarchy identical to the operation inheritance hierarchy shifts the burden of deciding what implementation to use from the calling code to the class hierarchy.
- *Sharing* – sharing of code using inheritance induces savings in code and more importantly conceptual simplicity by reducing the number of distinct cases that must be understood and analyzed.
- *Emphasis on object structure, not procedure structure* – stressing what an object *is*, rather than how is is *used*, makes the program more stable in the long run, since the features supplied by an object are much more stable than the way it is used as requirements evolve with time.
- *Synergy* – using identity, classification, polymorphism, and inheritance together results in usually cleaner, more general, and more robust program.

These abstract ideas will be made concrete by examples in the next section.

B. Fortran 90 implementation of object-orientation

In this section we describe the object-oriented construction of grid-related objects in Fortran 90. Fortran 90 keywords will be denoted as bold uppercase characters, and object-oriented concepts relevant to the discussion will be shown in italic characters. Before proceeding, we briefly consider the modeling of objects: The most basic components in the finite-difference electronic-structure code is the grid, which has the information of grid coordinates and number of grid panels along the $x-$,$y-$, and $z-$directions. We choose to use a uniform grid along each direction. In actual calculations, however, we employ only a localized region in real space to save the memory and enhance the computational efficiency. In addition, the grid is apparently constructed in a simulation box with basic information on the grid starting and finishing coordinates. So, we actually have a hierarchy of three grid-related physical objects whose two-dimensional representations as shown in Fig. 2. Below we present the implementations of this hierarchy of concepts using **TYPE**s of simulation box (**simbox**), grid (**grid**), and sub-grid (**subgrid**)[3].

□ **MODULE** for **TYPE simbox** and corresponding procedures

```
MODULE m_simbox
  IMPLICIT NONE
  PRIVATE
  PUBLIC :: simbox,new,display,...
  TYPE simbox
 ! Initial/final coordinates of the simulation box
 ! along x/y/z-dir.
 REAL :: xi,xf,yi,yf,zi,zf
  END TYPE simbox
  INTERFACE new
 MODULE PROCEDURE simbox_construct
  END INTERFACE
  INTERFACE display
 MODULE PROCEDURE simbox_print
  END INTERFACE
  ...
CONTAINS
  SUBROUTINE simbox_construct(x1,x2,y1,y2,z1,z2,box)
  ! Assign given initial/final coordinates of the simulation
  ! box to 'simbox' components.
REAL, INTENT(IN) :: x1,x2,y1,y2,z1,z2
```

[3]All the program listings in this paper have been simplified from original versions for clarity of presentation. Each module includes more subroutines, and double-precision has been used for real variables.

```
TYPE(simbox), INTENT(OUT) :: box
...
  END SUBROUTINE simbox_construct
  SUBROUTINE simbox_print(box,name)
  ! Print out simulation box information.
TYPE(simbox), INTENT(IN) :: box
CHARACTER*(*), INTENT(IN), OPTIONAL :: name
...
  END SUBROUTINE simbox_print
  ...
END MODULE m_simbox
```

□ **MODULE** for **TYPE grid** and corresponding procedures.

```
MODULE m_grid
  USE m_simbox, ONLY: simbox
  IMPLICIT NONE
  PRIVATE
  PUBLIC :: grid,new,delete,display,...
  TYPE grid
 ! Pointer to simulation box
 TYPE(simbox), POINTER :: pt_simbox
 ! Number of grid panels along x/y/z dir.
 INTEGER  :: nx,ny,nz
 ! Number of total grid points
 INTEGER  :: ngrid
 ! Grid spacings along x/y/z dir.
 REAL :: dx,dy,dz
 ! grid coordinates along x/y/z dir
 REAL, DIMENSION(:), POINTER :: xcrd,ycrd,zcrd
  END TYPE grid
  INTERFACE new
 MODULE PROCEDURE grid_construct
  END INTERFACE
  INTERFACE delete
 MODULE PROCEDURE grid_destruct
  END INTERFACE
  INTERFACE display
 MODULE PROCEDURE grid_print
  END INTERFACE
  ...
CONTAINS
  SUBROUTINE grid_construct(box,n1,n2,n3,grd)
```

```
    ! For the given simulation box, and the number of grid
    ! panels along each direction, assign/construct grid
    ! components.
  TYPE(simbox), INTENT(IN), TARGET :: box
  INTEGER,  INTENT(IN) :: n1,n2,n3
  TYPE(grid), INTENT(OUT) :: grd
  ...
  ! Simulation box coordinates
  grd%pt_simbox => box
  ...
  ! Grid coordinates generation.
  ALLOCATE(grd%xcrd(0:n1), ... STAT=ierr)
  IF(ierr/=0) ... ! Allocation error handling
  ...
    END SUBROUTINE grid_construct
    SUBROUTINE grid_destruct(grd)
  TYPE(grid) :: grd
  ...
  ! Nullify pointer to 'simbox'
  IF(ASSOCIATED(grd%pt_simbox)) NULLIFY(grd%pt_simbox)
  ! Deallocate 'xcrd','ycrd','zcrd'
  IF(ASSOCIATED(grd%xcrd)) THEN
     DEALLOCATE(grd%xcrd, STAT=ierr)
     IF(ierr/=0) ... ! Deallocation error handling
  ENDIF
  ...
    END SUBROUTINE grid_destruct
    SUBROUTINE grid_print(grd,name)
    ! Print out grid information.
  USE m_simbox, ONLY: display
  TYPE(grid), INTENT(IN) :: grd
  CHARACTER*(*), INTENT(IN), OPTIONAL :: name
  ...
  CALL display(box)
  ...
    END SUBROUTINE grid_print
    ...
  END MODULE m_grid
```

□ **MODULE** for **TYPE subgrid** and corresponding procedures.

```
  MODULE m_subgrid
    USE m_grid,   ONLY: grid
  IMPLICIT NONE
  PRIVATE
  PUBLIC :: subgrid,new,delete,...
  TYPE subgrid
 ! Pointer to 'grid'
 TYPE(grid), POINTER :: pt_grid
 ! Number of sub-grid points
 INTEGER :: nsubgrid
 ! local grid index number,
 ! 0 for outside of localized region.
 INTEGER, DIMENSION(:,:,:), POINTER :: index
  END TYPE subgrid
  INTERFACE new
 MODULE PROCEDURE subgrid_construct
  END INTERFACE
  INTERFACE delete
 MODULE PROCEDURE subgrid_destruct
  END INTERFACE
  ...
CONTAINS
  SUBROUTINE subgrid_construct(grd,sbgrd)
TYPE(grid), INTENT(IN), TARGET :: grd
TYPE(subgrid), INTENT(OUT) :: sbgrd
...
! Assign pointer to full grid.
sbgrd%pt_grid => grd
! Allocate subgrid index array and initialize
ALLOCATE(sbgrd%index(0:grd%nx,0:grd%ny,0:grd%nz), &
 STAT=ierr)
IF(ierr/=0) ! Deallocation error handling
...
  END SUBROUTINE subgrid_construct
  SUBROUTINE subgrid_destruct(sbgrd)
TYPE(subgrid) :: sbgrd
...
! Nullify pointer to 'grid', 'pt_grid'
IF(ASSOCIATED(sbgrd%pt_grid)) NULLIFY(sbgrd%pt_grid)
! Deallocate 'index' array of type 'subgrid' variable
IF(ASSOCIATED(sbgrd%index)) THEN
   DEALLOCATE(sbgrd%index,STAT=ierr)
   IF(ierr/=0) ! Deallocation error handling
ENDIF
  END SUBROUTINE subgrid_destruct
  ...
END MODULE m_subgrid
```

First, note that in each case we define a new **TYPE** which consists of corresponding variables, such as **xi,xf, etc.** for **simbox** and **nx,ny,nz, etc.** for **grid**. The ability to define derived **TYPE**s is a crucial ingredient of object-oriented code construction. [*abstraction*] By using these newly defined **TYPE**s, we construct/destroy corresponding variables together, and especially pass them to procedures (**FUNCTION**s and **SUBROUTINE**s) as a single argument, which enables the procedure interfaces to be simple and stable. [*identity*] We locate a **TYPE** definition in the corresponding **MODULE** to make it globally accessible. In addition, note that we hide the implementation details by first declaring all the entities in the **MODULE** as **PRIVATE** and list only exceptions as **PUBLIC** for *data hiding*[4]. [*encapsulation*] New **TYPE** definitions made **PUBLIC** to outside can be **USE**d in the calling routines by including the corresponding module, and individual components of the **TYPE** can be accessed by following the variable by a percentage sign % and the name of the component[5]. Note that, for enhanced safety, we use **ONLY** qualifier to access public entites in the **USE**d module.

Secondly, in all the **MODULE**s, we **CONTAIN** procedures which operate on the corresponding **TYPE** definition. Hence when employing each **MODULE**, the user will access the data structure and its behavior at the same time. [*classification*] Hence at this stage we have achieved the basic requirements for the "object-orientation": we organized program as collection of discrete objects that incorporate both data structure (**TYPE**) and behavior (procedures attached to the **TYPE** by being **CONTAIN**ed).

Next, note that, when we **CONTAIN** procedures, we employ **MODULE PROCEDURE** statement to give them generic names, and make those generic names (instead of original names of the procedures) be accessible from the outside by giving them a **PUBLIC** attribute. In doing so, we can give different procedures a single generic name. For example, we use the same generic name **new** for different procedures of **TYPE**s, **simbox** (**simbox_construct**), **grid** (**grid_construct**), and **subgrid** (**subgrid_construct**).[*polymorphism*[6]]

Finally, in the **MODULE m_grid**, **TYPE grid** inherits the **simbox** information, and again this **grid** information (including that of **simbox**) is inherited to **TYPE subgrid**

[4] One can list all the entities in each **MODULE** as **PRIVATE** or **PUBLIC**, or make the default **PUBLIC** and then only list exceptions as **PRIVATE**. However, for the purpose of *data hiding*, the current form is strongly recommended [14].

[5] One can even hide the data components of a derived **TYPE** to the outside of the **MODULE** by preceding the first component declaration in the derived **TYPE** definition by **PRIVATE** attribute. In that case, to access the components, it is required to write procedures which manipulate and return the components and include them in the same **MODULE**.

[6] To be more specific, this is *static polymorphism*. A good discussion on how to implement a *run-time polymorphism* can be found in Ref. [18].

in **MODULE m_subgrid**. [*inheritance*] So, **TYPE grid** variables will contain information on the simulation box (**xi,xf, etc**), and **TYPE subgrid** variables will contain the information on the **grid** (**nx,ny,nz, etc.**) and the **simbox** in which the **grid** has been constructed. Note that for the implementation of this hierarchy structure, we have used **POINTER**s. In Fortran 90, to avoid execution efficiency degradation, all objects to which a **POINTER** may point should have a **TARGET** attribute. Hence, input variables (**INTENT** attribute **IN**) **box** in the procedure **grid_construct** (generic name **new**) of the **MODULE m_grid** and **grd** in the procedure **subgrid_construct** (generic name **new**) of the **MODULE m_subgrid** have the **TARGET** attribute. **POINTER**s have been associated with **TARGET**s by **POINTER** assignment statements:

pointer => *target*

in the procedures **grid_construct** and **subgrid_construct**. In addition inheriting data, it is also possible for a procedure to inherit another procedure. In our example, a procedure **grid_print** in the **MODULE m_grid** inherits (by **USE** statement) another procedure **simbox_print** (generic name **display**) located in the **MODULE m_simbox**. Again, note that we give a generic name **display** to both **grid_print** and **simbox_print**.

Now we comment on additional Fortran 90 language features related with our examples. First, we do not use a **MODULE** as a storage place of global variables, although it is possible to do so, because we found it is rather clumsy and risky for the large-scale coding due to the problem of global storage similar to that arises in the usage of COMMON block in Fortran 77. **MODULE** is exclusively used as a place for **TYPE** definitions and corresponding procedures. Next, since **ALLOCATABLE** arrays cannot be used in derived **TYPE** definitions, we use instead **POINTER**s (see **grid_construct** and **subgrid_construct**). Again, in this case of **POINTER** usage, we arrange the allocation to occur in a well-defined corresponding procedure contained in the same module, which makes the usage of the derived **TYPE** more safe and robust. Note that pointer disassociation (**NULLIFY**) and deallocation (**DEALLOCATE**) are also handled in a similar way (see **grid_destruct** and pwd**subgrid_detruct**).

Before closing this section, we summarize the strategy of the modeling of a new (or the remodeling of a present) large-scale code in Fortran 90 with object-oriented concepts, which we found useful.

1. Identify an object and define the corresponding variables as a **TYPE**. These variables are typically global variables, or frequently passed variables from the main program to procedures, used together in a procedure in the conventional non-object-oriented programming style. Be careful on the hierarchy (dependence) of the objects.

2. Construct (identify) subroutines closely related with the **TYPE**. (Re)arrange the procedure interfaces (and contents if necessary) using the **TYPE** definition.

3. Define a **MODULE** corresponding to the **TYPE**, and include the **TYPE** definition from step 1 and **CONTAIN** procedures identified in step 2 in the **MODULE**, and give them generic names.

4. Make the **TYPE** definition and generic names of procedures **PUBLIC**.

Remind that the initial stage takes most of the time in the object-oriented approach, which is especially true in the remodeling of an existing code, since the remaining process is mostly changing interfaces and variable names in existing procedures and including them in **MODULE**s (assuming that the previous code is well-designed, hence the restructuring is straightforward). It should be also emphasized that we find this process is actually beneficial for doing physics itself, in that the programmer (usually the physicist herself or himself) has to consider (reconsider in the case of remodeling of a present code) and identify carefully the structures embedded in the problem under consideration, hence results in making one focus more on the physical pictures. We refer the reader to Ref. [13] for further discussion of the benefits of object-oriented programming.

IV. OBJECT-ORIENTED IMPLEMENTATION OF MULTIGRID METHODS

A. Multigrid method

We first briefly review the theory of the multigrid method [9–12]. The fundamental idea behind all multigrid methods is to combine computations done on different scales, based on the observation that many iterative methods tend to reduce the high-frequency (i.e. oscillatory) components of the error rapidly but reduce the low-frequency (i.e. smooth) components of the error much slowly. Since the notions of smooth or oscillatory components of the error are relative to the mesh on which the solution is defined, in particular, a component that appears smooth on a fine grid may appear oscillatory on a coarse grid, we can naturally think of using coarser grid to reduce this (now oscillatory) error and interpolate back to the fine grid. The key to the successful implementation of a multigrid method is the choice of grids of different scales, the strategy to proceed through them, and how we move objects among them. Broadly the multigrid algorithms can be divided into two basic categories [12]:

1. Correction methods − start at the finest level, and use the coarser levels solely to compute a correction which is added to the approximate solution on the finest level.
2. Nested iteration methods − generate initial guesses on coarser levels and frequently reuse coarser levels for corrections also.

Specific examples of these correction path and nested iteration path are shown in Fig 3-(a) and (b). Roughly speaking, when a good initial guess is available, a correction algorithm can be used, otherwise we should use a nested iteration scheme [12].

B. One-way multigrid method in electronic-structure calculations and its implementation

Multigrid is a quite general concept, and apparently the choice of a specific algorithm depends on the nature of the problem under consideration. We have recently demonstrated that the introduction of a simple one-way multigrid method (Fig. 3-b) greatly improves the efficiency of real-space electronic-structure calculations based on the iterative solution of KS equations [8]. The motivation of our work was based on the observation that the most time-consuming part of the self-consistent electronic-structure calculations described in Sec. II is the iterative solution of KS equations [22,23]. The sources of this computation bottleneck can be traced to broadly two (but closely related) aspects of self-consistent iterative diagonalization schemes. First of all, in general we do not have a good initial guess of wave functions, which generate density, and hence $V_{\mathrm{H}}(\mathbf{r})$ and $V_{\mathrm{xc},\sigma}(\mathbf{r})$ in Eq. (1). So initial several self-consistency steps will be used to obtain solutions of biased Hamiltonians, although they tend to be the most time-consuming part. Secondly, in single iterative solution of KS equations, a direct application of a relaxation method on the fine grid has trouble in damping out the long-ranged or slowly varying error components in the orbitals. This can be understood by the usual spectral analysis of relaxation scheme [11], or considering that the nonlocal Laplacian operation on a fine grid is physically short-ranged.

Hence, for our purpose, we seek a multigrid procedure which generates a good initial guess for the finest grid calculation and effectively removes long-range error components of wave functions in the solution of KS equations [Eq. (1)]. While an efficient interpolation/projection scheme is a crucial ingredient of any successful application of multigrid method, we note that it can be also time-consuming and tricky part since in our case we need to transfer a large number of wave functions which should observe the orthonormality conditions. Hence our strategy, which is the characteristic of the scheme, is to minimize the number of data transfer between different grid levels, and employ an accurate interpolation method which is very accurate and allow us to use even a noninteger ratio of grid spacings: The calculation starts from the coarsest grid $2h$, and in each grid-level calculation, Eqs. (1) and (4) are solved self-consistently as in the usual single-level algorithm shown in Fig. 1. After each self-consistent calculation on a coarse grid, only wave functions are interpolated to the next fine grid, and another set of self-consistent calculation is performed. Since that the interpolated wave functions usually do not satisfy the orthonormality condition any more, we take an extra Gram-Schmidt orthogonalization process after each orbital interpolation. Hence we have $n-1$ interpolations and Gram-Schmidt orthogonalization processes for the $n-$level multigrid calculations. For the interpolation, we used a three-dimensional piecewise polynomial interpolation with a tensor product of one-dimensional B-splines as the interpolating function [10,24]. A piecewise cubic polynomials have been taken as B-splines. In Fig. 4, we summarize the computational procedure for the case of three grids, $2h$, $1.5h$, and h. We refer the reader to Ref. [8] for further discussion of the method and the comparison with other multigrid schemes [2,5,6].

Now we describe the implementation of the one-way multigrid algorithm of Fig. 4 using the object-oriented components constructed in Sec. III B. The main part of the code is shown below.

```
  SUBROUTINE mg_ks_dft(box,ng,...)
USE m_simbox, ONLY: simbox
USE m_grid, ONLY: grid,new,display,delete
USE m_subgrid, ONLY: subgrid,new,delete
USE m_wavefunction, ONLY: wavefunction,new,delete
...
IMPLICIT NONE
TYPE(simbox), INTENT(IN) :: box ! Simulation box information
INTEGER, INTENT(IN) :: ng(3) ! Number of grid panels
 ! at the finest grid
```

```
...
! Local variables
TYPE(grid), ALLOCATABLE, DIMENSION(:) :: grd
TYPE(subgrid), ALLOCATABLE, DIMENSION(:) :: sbgrd
TYPE(wavefunction), ALLOCATABLE, DIMENSION(:), TARGET :: wf
...

! Allocatable arrays
! 'nlevel' is number of multigrid level
ALLOCATE(grd(nlevel),sbgrd(nlevel),wf(nlevel),STAT=ierr)
IF(ierr/=0) ...  ! Error handling
...
! One-way multigrid loop
DO ilvl = 1,nlevel

   ! Assign number of grid panels for each multigrid level
   ! 'ratio' is grid spacing ratio with finest grid as 1
   n1 = NINT(ng(1)/ratio(ilvl))
   n2 = NINT(ng(2)/ratio(ilvl))
   n3 = NINT(ng(3)/ratio(ilvl))

   ! Generate grid for level='ilvl'
   CALL new(box,n1,n2,n3,grd(ilvl))
   CALL display(grd(ilvl))

   ! Generate subgrid for level='ilvl'
   CAL1 new(grd(ilvl),sbgrd(ilvl))

   ! Effective region allocation in the subgrid
   ...

   ! Generate wavefunctions
   CALL new(...,wf(ilvl))

   ! If level>1, interpolate w.f.(ilvl-1) => w.f.(ilvl)
   IF(ilvl>=2) THEN

  ! Interpolate: wf(ilvl-1) => wf(ilvl)
  ...
  ! and orthonormalize them
  ...

  ! Destruct: grid(ilvl-1), subgrid(ilvl-1), wf(ilvl-1).
  CALL delete(grd(ilvl-1))
  CALL delete(sbgrd(ilvl-1))
  CALL delete(wf(ilvl-1))
   ENDIF

   ! Adjust calculation parameters according to grid level.
   ...

   ! Solve KS-equations for level='ilvl'
   CALL ksdft(...)

   ! Remove grid(ilvl), subgrid(ilvl), wf(ilvl)
   ! at the last MG step
   IF(ilvl == nlevel) THEN
  CALL delete(grd(ilvl))
  CALL delete(sbgrd(ilvl))
  CALL delete(wf(ilvl))
   ENDIF

END DO

! Deallocate arrays
DEALLOCATE(grd,sbgrd,wf)
...

  END SUBROUTINE mg_ks_dft
```

The simplification of the coding induced by object-oriented programming style should be obvious in this example. Since the simulation box is assigned only once in the current method, it has been generated once in the higher level and passed as an input variable, and only grids and sub-grids built in the simulation box have been **ALLOCATE**d for the number of multigrid levels[7]. In addition to the objects we described in Sec. III B, we use **TYPE wavefunction** which has not been described but has been constructed in a similar way as others. Again, note that we use the same generic names of procedures for different **TYPE**s, **new** and **delete**. It should be also noted that actual allocatable arrays in **TYPE grid** (**xcrd, etc.**), **subgrid** (**index**), and most importantly **wavefunction** whose size is (number of grid points) $\times$ (number of states) $\times$ (number of spins) are only **ALLOCATE**d when they are required, and those arrays are **DEALLOCATE**d as soon as they become unnecessary. These processes are elegantly handled by **new** and **delete** calls.

[7]However, **simbox** objects can be also frequently generated if we perform adaptive mesh refinement type calculations.

C. Performance test

For the analysis of performance enhancement due to our multigrid method, we reconsider a quasi-two-dimensional quantum dot model that has been employed in Ref. [8], in which a 20-electron quantum dot has been studied with one-level (h), two-level ($2h$ and h), and three-level ($2h$, $1.5h$, and h) methods. Here, a more detailed analysis of performance is provided, with varying number of electrons up to 24 and additional four-level ($4h$, $2h$, $1.5h$, and h) calculations. While calculations were performed on the entire simulation box with the original code in Ref. [8] , here we employ the newly designed object-oriented code which use only the grid points inside of a spherical region with a radius 8.0 a_B^*. Two calculations are further different in simulation parameters.

Quantum dot in GaAs host material (dielectric constant $\epsilon = 12.9$, effective mass $m^* = 0.067m_e$) is modeled by an anisotropic parabolic confinement potential $V_{\text{ext}}(\mathbf{r}) = \frac{1}{2}\omega_x^2 x^2 + \frac{1}{2}\omega_y^2 y^2 + \frac{1}{2}\omega_z^2 z^2$, in which the z-axis is taken as the dot growth direction. As in Ref. [8], we use the confinement energies $\omega_x = \omega_y = 5$ meV, and $\omega_z = 45$ meV. Our calculations are based on the effective mass approximation, and rescaled length and energy units are respectively a_B^* $= 101.88$ $\mathbf{r}A$ and 10.96 meV. Uniform grid spacing $h = 0.3a_B^*$ with box size $18 \times 18 \times 18 {a_B^*}^3$ have been used. Incorporation of the spherical local region results in the usage of only about 35% of total number of grids, hence the number of grid points involved in the calculations is 1.2×10^3 for grid $4h$, 1.0×10^4 for grid $2h$, 2.4×10^4 for grid $1.5h$, and 7.9×10^5 for grid h. Finite-difference order N [Eq. (5)] for the solution of the KS equations and Poisson equation are chosen such that the range of the physical coverage is approximately same, so $N = 5$ for h, $N = 3$ for $1.5h$ and $2h$, and $N = 1$ for $4h$. Noninteracting eigenstates (Hermite polynomials) are used as an initial guess for the coarsest grid calculation. Spin-unpolarized calculations have been performed for the simplicity of performance comparison, although the spin-polarized scheme should be employed to observe possible spin-polarized states and the corresponding Hund's rule [3].

We first show the CPU times of self-consistent iterations in 1-, 2-, 3-, and 4-level 24-electron calculations in Fig. 6 to contrast the characteristics of self-consistent calculations in the conventional 1-level and multigrid methods. The horizontal axis stands for the self-consistency iteration index, while the vertical axis is the required computer time for a given iteration step. Interpolation and orthonormalization steps in multi-level calculations are indicated by downward arrows. While the multigrid calculations requires more number of self-consistent iterations in general, they are mostly performed in the coarsest grid, and at the finest grid level h we only need two or three iterations, which demonstrates that coarse grid calculations provide a good initial solution for the finest grid h calculation and results in significant time saving.

In Fig. 5, we compare the performance of different multigrid strategies for different number of electrons. Note that in general the use of multigrid improves the computation speed, and moreover its efficiency increases with the system size. Computation speed-up defined as (CPU time for 1-level calculation)/(CPU time for n-level calculation) amounts to more than 7 for the 24 electron case with 4-level method. Second, while the 3- and 4-level computations are usually better than the 2-level one, its specific performance varies with the number of electrons. The rule of thumb is that 4-level method should be used for the electron number larger than 20, otherwise 3-level is sufficient.

V. CONCLUSIONS

In the modern computation era when the increase of computational capability increases almost exponentially with time, it is clear that physicists can attack more ambitious problems requiring more challenging large-scale computations. However, with the growth of the size of the problem, typically the complexity of the problem itself, hence the complication of the code also increases. Object-oriented methodology can be a valuable solution to this problem of complexity of modern scientific computations, and, in this paper, we showed one example of the application of the object-orientation methodology to the large-scale code implementation in Fortran 90. Specifically, we treated a real-space grid-based electronic-structure program which solves the KS and Poisson equations self-consistently, and especially explained how we have implemented the one-way multigrid method we have recently proposed [8] using the object-oriented techniques. According to our experience, we believe that it pays to write a scientific program in object-oriented fashion in the long run, and further the cost we have to pay is minimal compared with its benefits even when using a non-object-oriented language Fortran 90.

ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation under Grant No. DMR 9802373 (Y.-H.K. and R.M.M). Computations were performed in the Material Research Laboratory Center for Computation.

REFERENCES

[1] J. R. Chelikowsky, N. Troullier, and Y. Saad, *Phys. Rev. Lett* **72** (1994) 1240; J. R. Chelikowsky, N. Troullier, K. Wu, and Y. Saad, *Phys. Rev.* **B50** (1994) 11355.

[2] E. L. Briggs, D. J. Sullivan, and J. Bernholc, *Phys. Rev.* **B52** (1995) R5471; *Phys. Rev.* **B54** (1996) 14362.

[3] I.-H. Lee, V. Rao, R. M. Martin, and J.-P. Leburton, *Phys. Rev.* **B57** (1998) 9035; I.-H. Lee, K.-H. Ahn, Y.-H. Kim, R. M. Martin, and J.-P. Leburton, *Phys. Rev.* **B** (1999), to be published.

[4] Y.-H. Kim, I.-H. Lee, and R. M. Martin, in *Stochastic Dynamics and Pattern Formation in Biological and Complex Systems*, edited by S. Kim, K. Lee, T.K. Lim and W. Sung (AIP, 1999); Y.-H. Kim, M. Städele, and R. M. Martin, *Phys. Rev.* **A60** (1999) 3633.

[5] F. Ancilotto, P. Blandin, and F. Toigo, *Phys. Rev.* **B59** (1999) 7868.

[6] J. Wang and T. Beck, preprint cond-mat/9905422 and references therein.

[7] S. Goedecker, *Rev. Mod. Phys.* **71** (1999) 1085.

[8] I.-H. Lee, Y.-H. Kim, and R. M. Martin, *Phys. Rev.* **B** (1999), submitted.

[9] M.T. Heath, *Scientific Computing: An Introductory Survey.* (Mcgraw-Hill, New York, 1997).

[10] W.H. Press, S.A. Teukkolsky, W.T. Vetterling, B. Flannery, *Numerical Recipies in Fortran, 2nd edition.* (Cambridge University, Cambridge, England, 1992).

[11] W. Briggs, *A Multigrid Tutorial.* (SIAM, Philadelphia, 1987).

[12] C.C. Douglas, *IEEE Computational Science & Engineering*, **Winter** (1996) 55.

[13] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* (Prentice-Hall, Englewood Cliffs, NJ, 1991).

[14] T.M.R. Ellis, I.R. Philips, T.M. Lahey, *Fortran 90 Programming.* (Addison-Wesley, Wokingham, England, 1994).

[15] V.K. Decyk, C.D. Norton, and B.K. Szymanski, *ACM Fortran Forum* **16** (1997) 13.

[16] J.R. Cary, S.G. Shasharina, J.C. Cummings, J.V.W. Reynders, and P.J. Hinker, *Comput. Phys. Comm.* **105** (1997) 20.

[17] M.G. Gray and R.M. Roberts, *Comput. Phys.* **11** (1997) 355.

[18] V.K. Decyk, C.D. Norton, and B.K. Szymanski, *Comput. Phys. Comm.* **115** (1998) 9.

[19] P.F. Dubois, *Sci. Programming* **1** (1999) 7; ftp-icf.llnl.gov/pub/OBF90.

[20] P. Hohenberg and W. Kohn, *Phys. Rev.* **136** (1964) B864; W. Kohn and L. J. Sham, *Phys. Rev.* **140** (1965) A1133.

[21] B. Fornberg and D. Sloan, in *Acta Numerica 1994*, A. Iserles ed., (Cambridge University Press, Cambridge, 1994) p. 203.

[22] M. P. Teter, M. C. Payne, and D. C. Allan, *Phys. Rev.* **B40** (1989) 12255; M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos, *Rev. Mod. Phys.* **64** (1992) 1045.

[23] D. M. Bylander, L. Kleinman, and S. Lee, *Phys. Rev.* **B42** (1990) 1394.

[24] C. de Boor, *A Practical Guide to Splines, 2nd edition.* (Springer-Verlag, New York, 1984).

FIGURES

FIG. 1. Flowchart of the current higher-order finite-difference electronic-structure calculations based on the iterative CG diagonalization. Only the dotted parts are repeated in the higher-level multigrid calculations shown in Fig. 4.

FIG. 2. Two-dimensional representation of the hierarchy of three physical objects for grid-based calculations: (a) simulation box, (b) grid, and (c) sub-grid. Only filled circles in (c) are actually used for computations.

FIG. 3. Examples of multigrid algorithmic flow: (a) correction path, and (b) nested iteration path. Level 3 has the finest grid, level 1 the coarsest; computation flows from left to right.

FIG. 4. Schematic diagram of the present one-way multigrid algorithm for the case of three-level ($2h$, $1.5h$, and h) calculations. The calculation starts at the coarsest level (level 1, $2h$) at the bottom, and ends at the finest grid (level 3, h) at the top. At level 2 and 3, only the dotted parts of the self-consistent calculation in Fig. 1 are performed. Orbital interpolation and orthogonalization step is taken after each coarse grid (level 1 and 2) calculation.

FIG. 5. CPU time vs. self-consistent iteration number of twenty-four-electron quantum dot calculations in (a) one-level (h), (b) two-level ($2h$ and h), (c) three-level ($2h$, $1.5h$, and h), and (d) four-level ($4h$, $2h$, $1.5h$, and h) schemes. Downward arrows in (b), (c), and (d) indicate interpolation-orthonormalization steps. Total computation time is (a) 59.5, (b) 12.9, (c) 11.0, and (d) 8.3 minutes. Calculations are performed on DEC alpha 500au workstations.

FIG. 6. Comparison of the computational efficiency enhancement in $n-$level one-way multigrid methods, where n is 2 ($2h$ and h), 3 ($2h$, $1.5h$, and h), and 4 ($4h$, $2h$, $1.5h$, and h), for electron number 8, 12, 16, 20, and 24. Computation speed-up has been defined as (CPU time for 1-level calculation)/(CPU time for n-level calculation). Total computation time of 1-level calculation has been denoted in minutes for each electron number. Calculations are performed on DEC alpha 500au workstations.

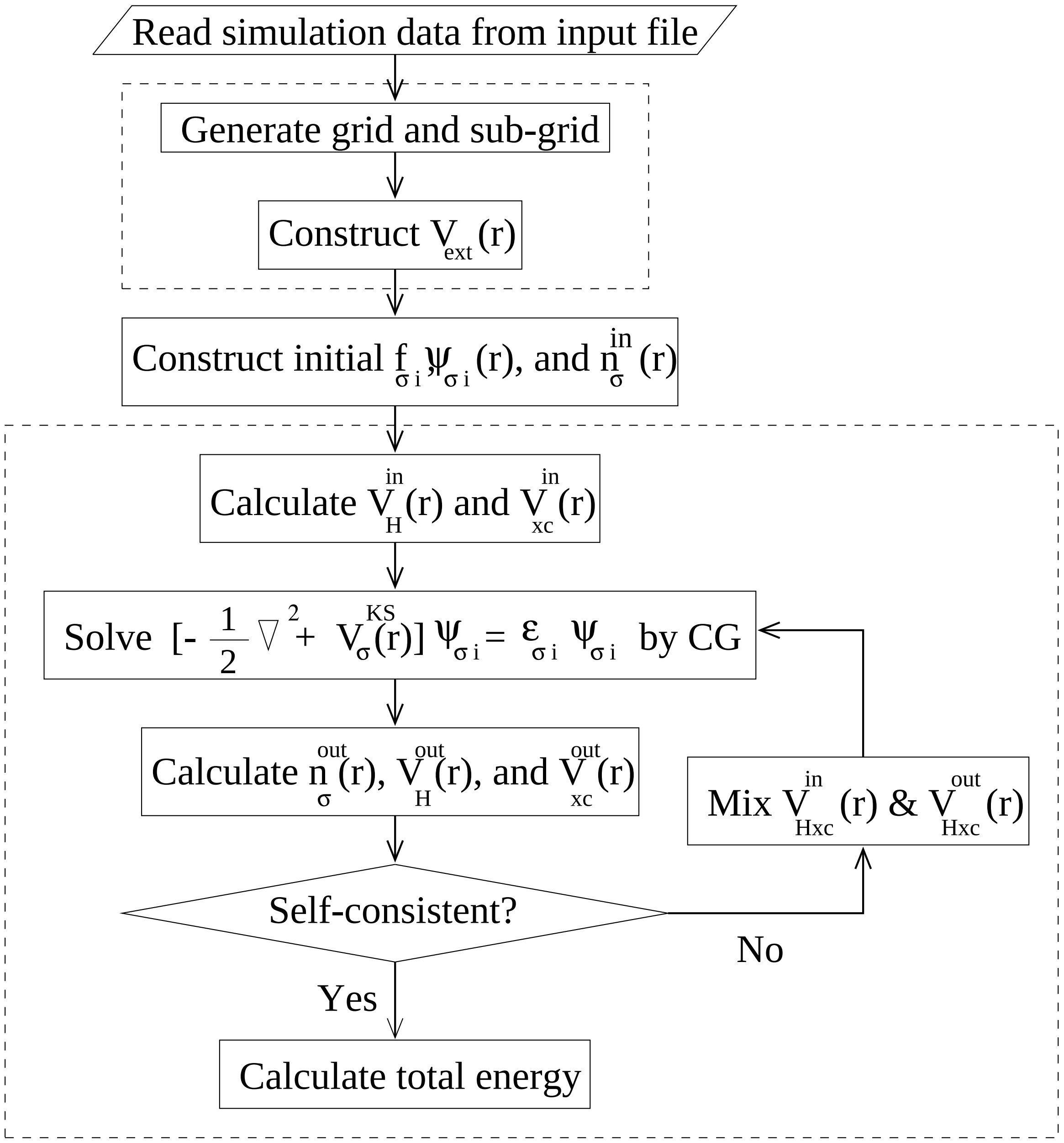
Read simulation data from input file
Generate grid and sub-grid
Construct $V_{ext}(r)$
Construct initial $f_{\sigma i}$, $\psi_{\sigma i}(r)$, and $n^{in}_{\sigma}(r)$
Calculate $V^{in}_{H}(r)$ and $V^{in}_{xc}(r)$
Solve $[-\frac{1}{2}\nabla^2 + V^{KS}_{\sigma}(r)]\psi_{\sigma i} = \varepsilon_{\sigma i}\psi_{\sigma i}$ by CG
Calculate $n^{out}_{\sigma}(r)$, $V^{out}_{H}(r)$, and $V^{out}_{xc}(r)$
Mix $V^{in}_{Hxc}(r)$ & $V^{out}_{Hxc}(r)$
Self-consistent?
No
Yes
Calculate total energy

Fig. 1

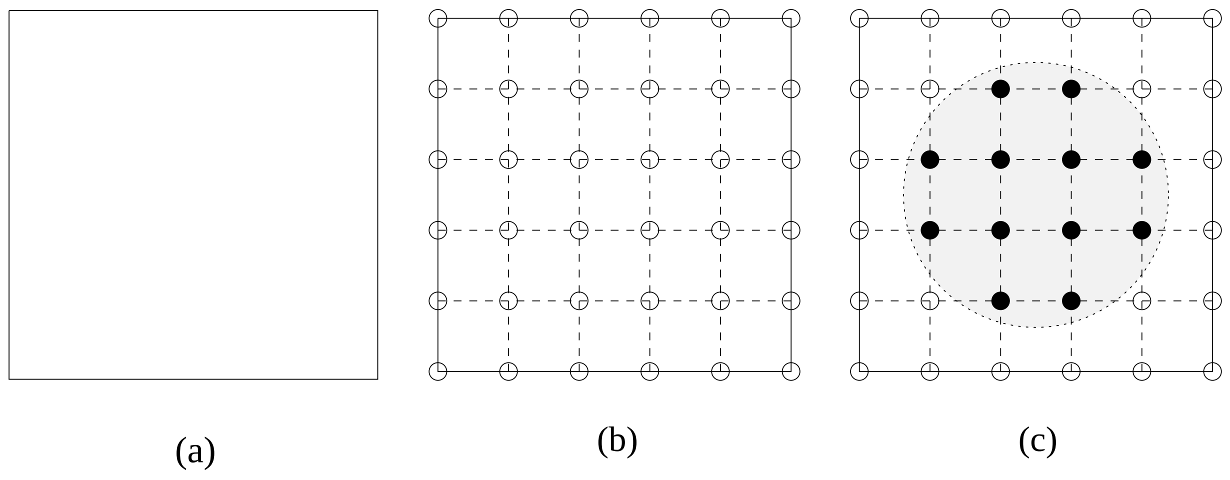

Fig. 2

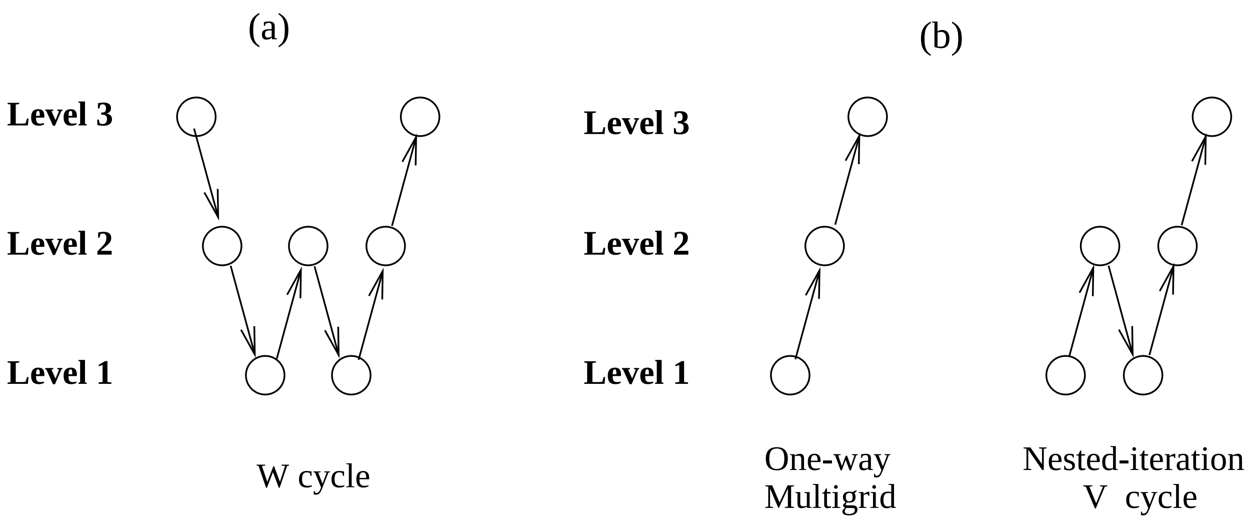
(a)
Level 3
Level 2
Level 1
W cycle
(b)
Level 3
Level 2
Level 1
One-way Multigrid
Nested-iteration V cycle

Fig. 3

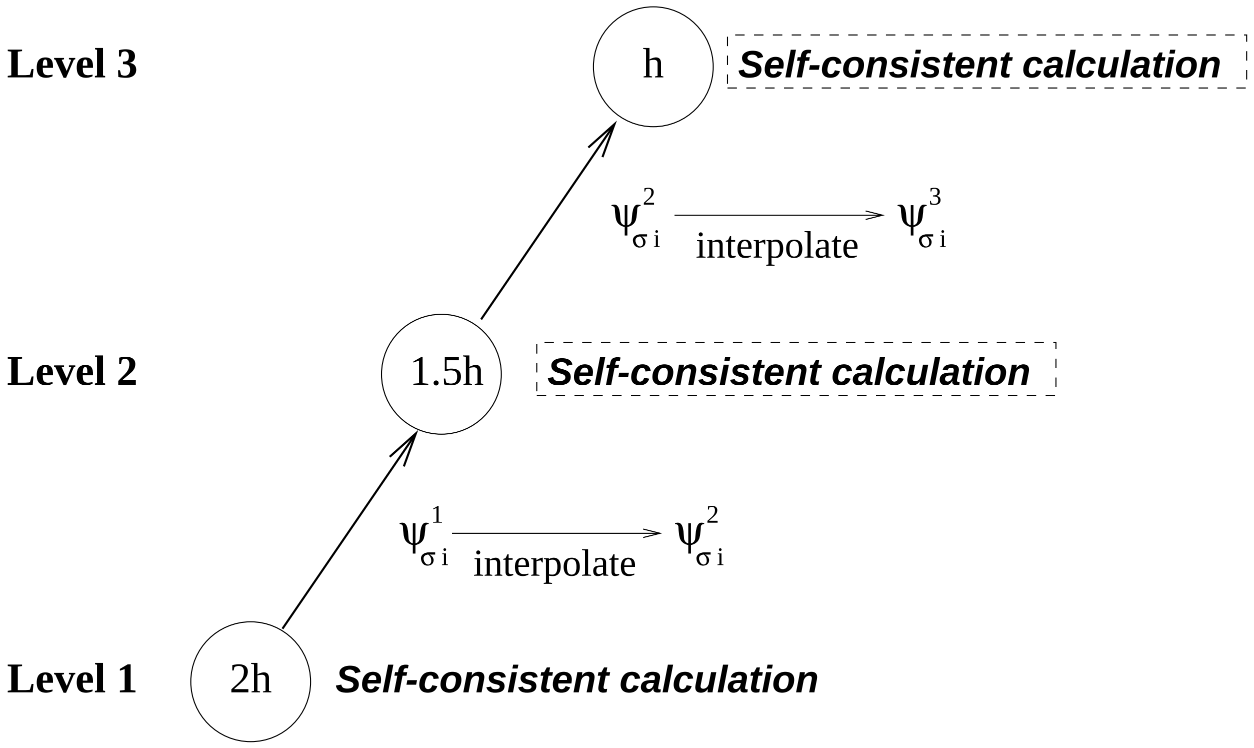
Level 3
h
Self-consistent calculation
$\psi_{\sigma i}^{2}$
interpolate
$\psi_{\sigma i}^{3}$
Level 2
1.5h
Self-consistent calculation
$\psi_{\sigma i}^{1}$
interpolate
$\psi_{\sigma i}^{2}$
Level 1
2h
Self-consistent calculation

Fig. 4

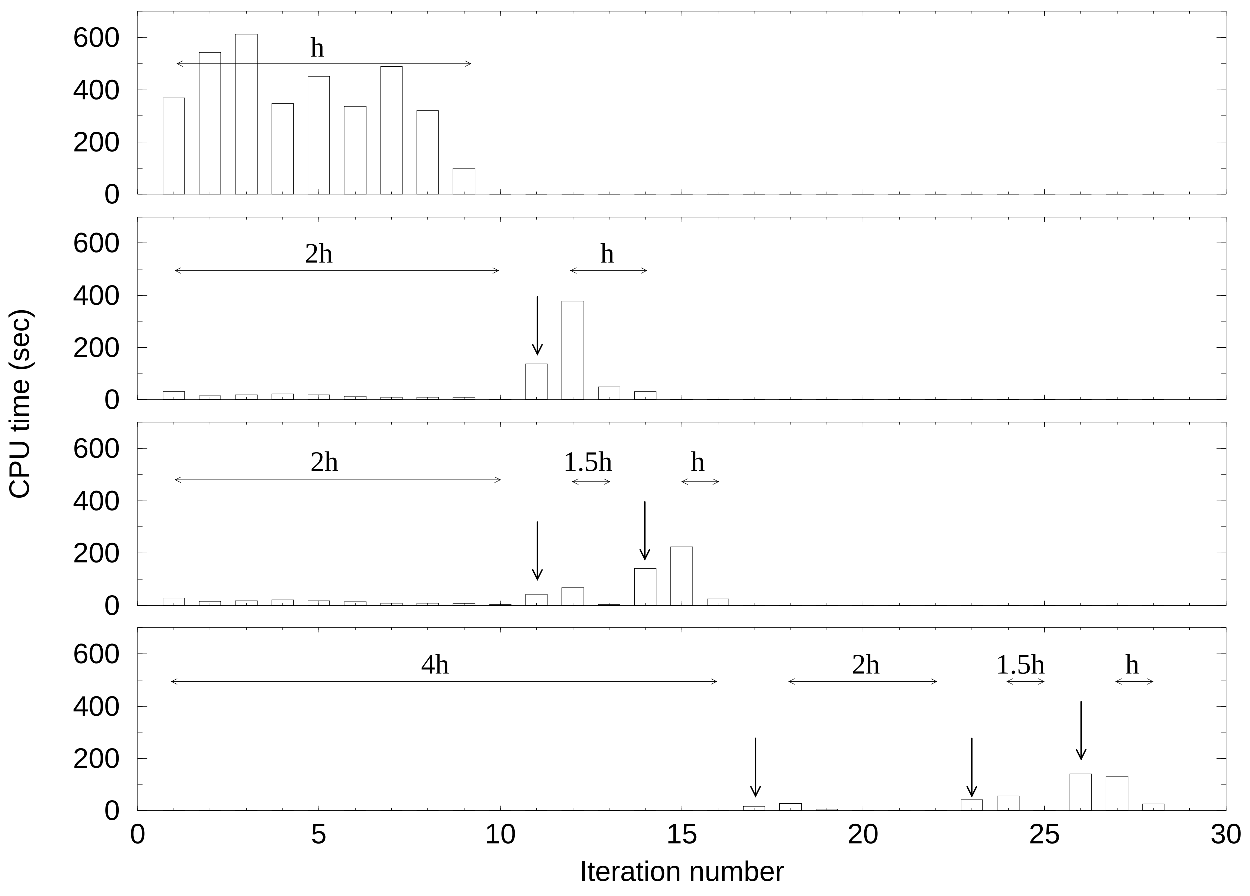
h
2h
h
2h
1.5h
h
4h
2h
1.5h
h
CPU time (sec)
Iteration number
0
200
400
600
0
5
10
15
20
25
30

Fig. 5

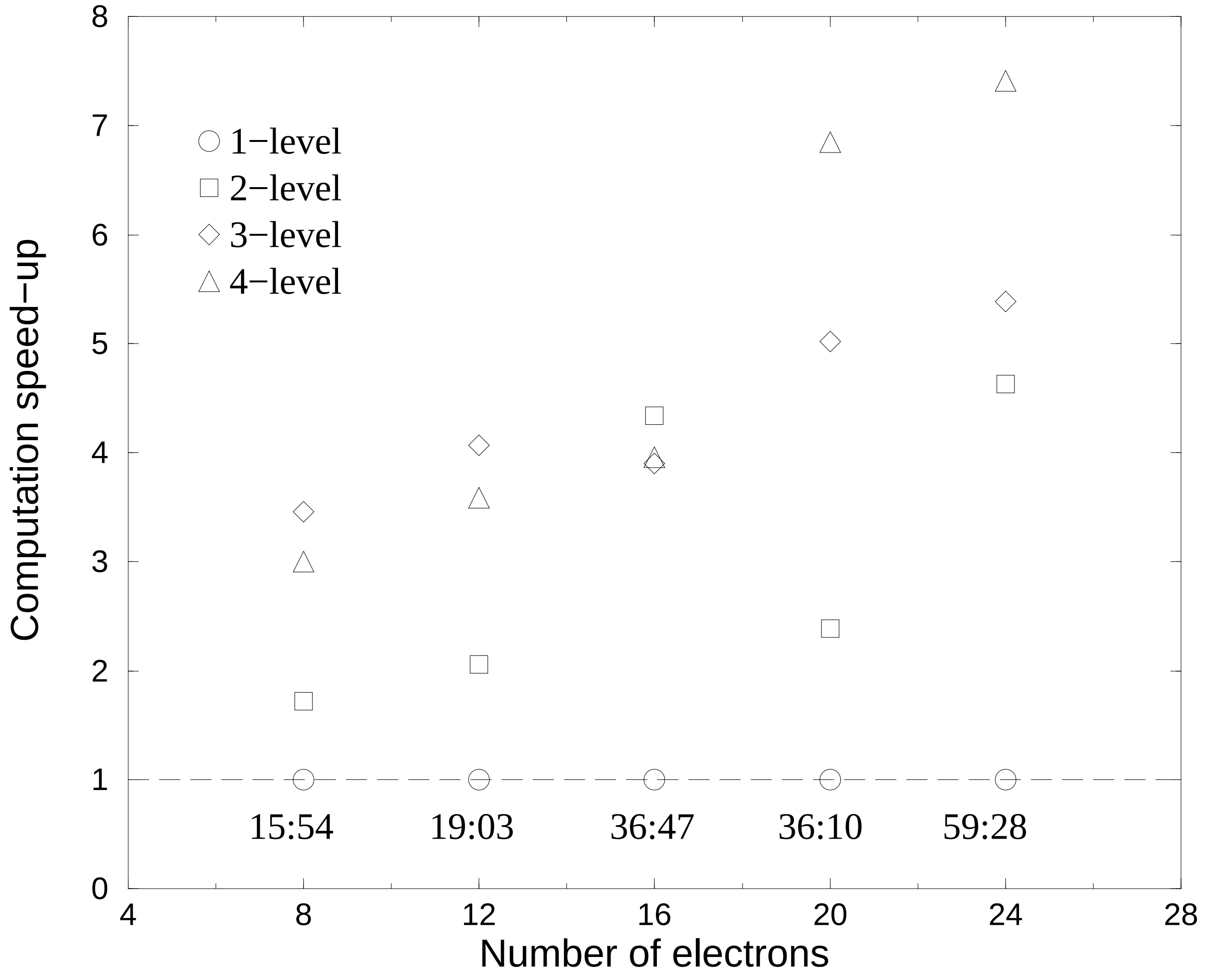
8
7
6
5
4
3
2
1
0
4
8
12
16
20
24
28
Computation speed−up
Number of electrons
○ 1−level
□ 2−level
◇ 3−level
△ 4−level
15:54
19:03
36:47
36:10
59:28

Fig. 6